Práctica 2: Pacman 2.0

Curso 2020-2021. Tecnología de la Programación de Videojuegos 1. UCM

Fecha de entrega: 18 de diciembre de 2020

El objetivo fundamental de esta práctica es introducir el uso de la herencia y el polimorfismo en la programación de videojuegos mediante C++/SDL. Para ello, partiremos de la práctica anterior y desarrollaremos una serie de extensiones que se explican a continuación. En cuanto a funcionalidad, el juego presenta las siguientes modificaciones/extensiones:

1. Cuando se acaba con toda la comida del mapa, o bien al llegar a determinada puntuación, se pasa al siguiente nivel (otro mapa diferente que se carga del fichero correspondiente). Se deja libertad en cuanto a la mecánica del juego en este sentido y en cuanto a cuándo se considera que el juego se acaba.

2. Ahora hay dos clases de fantasmas: los fantasmas simples (igual que los de la práctica 1), de colores rojo, amarillo, rosa o azúl, y los fantasmas *inteligentes* o *evolucionados*, de color morado (representados en los ficheros con un 4). Éstos últimos nacerán (con un tamaño más pequeño), al ser adultos (ya con tamaño normal) se reproducirán (si se dan las condiciones) y morirán. Siendo adultos se moverán de manera agresiva tratando de acercarse al pacman, y se reproducirán si se cruzan con otro fantasma evolucionado adulto o normal, y hay un espacio libre al lado de alguno de los padres, dando lugar a un nuevo fantasma que aparecerá pegado a sus padres, y que será evolucionado si ambos padres lo son, o normal en otro caso. Al llegar a cierta edad los fantasmas evolucionados morirán y su cuerpo quedará inerte en el sitio hasta pasado cierto tiempo en el que desaparecerán de la escena.

3. Como puedes observar en el vídeo de demostración el movimiento del pacman y fantasmas es ahora continuo (a nivel de píxeles y no de coordenadas del mapa como antes).

4. Las partidas pueden guardarse y cargarse: Al iniciar el juego aparecerá un menú con dos opciones, jugar y cargar partida. En caso de seleccionarse la opción de cargar partida, se introducirá a continuación el código numérico de la partida y se cargará la partida a partir del fichero correspondiente (en caso de encontrarse). De manera análoga, mientras se está jugando, si se pulsa la tecla *s* seguida de un código numérico y seguida de *intro*, la partida se guardará con el código numérico introducido.

Detalles de implementación

Cambio de nivel

Como se ha indicado, cuando se acaba con toda la comida del mapa o bien al llegar a determinada puntuación, el juego pasa al siguiente nivel. El juego incluye por tanto un nuevo atributo que indica el nivel actual. Éste se utiliza para formar el nombre del fichero del siguiente nivel, el cual tiene la forma `levelX.pac` siendo X el nivel correspondiente. Para ello puedes hacer uso de un flujo de tipo `stringstream`, que permite manejar *strings* como flujos de texto y por tanto aprovechar las funcionalidades para insertar y extraer datos en flujos de texto. Al pasar de nivel se deben destruir todos los objetos de la escena y volver a crearse a partir de los datos del fichero, excepto el objeto marcador (en caso de existir).

Diseño de clases

A continuación se indican las nuevas clases y métodos que debes implementar obligatoriamente y las principales modificaciones respecto a las clases de la práctica 1. Deberás implementar además los métodos (y posiblemente las clases) adicionales que consideres necesario para mejorar la claridad y reusabilidad del código. Como norma general, cada clase se corresponderá con la definición de un módulo C++ con sus correspondientes ficheros .h y .cpp (excepto en clases sin nada de implementación, que solo tendrán fichero .h).

Clases GameObject, GameCharacter y jerarquía de objetos del juego: La clase raíz de la jerarquía es la clase abstracta `GameObject`, que declara la funcionalidad común a todo objeto de una aplicación/juego SDL (método abstracto `render`, métodos virtuales `update` y `getDestRect`, y destructora virtual). Declara también atributos para la posición del objeto (tipo `Point2D`), la anchura, la altura y un puntero al juego. La clase `GameCharacter`, también abstracta (en este caso por no tener un constructor público), hereda de `GameObject` y declara, y también define de forma genérica toda aquella funcionalidad común a pacman y fantasmas (métodos `render` y `saveToFile`). Incluye atributos para la posición inicial, la dirección/velocidad de movimiento, un puntero a la textura del objeto, las coordenadas del frame de la textura, y finalmente, un iterador apuntando a la posición del objeto en la lista de objetos del juego. Define una constructora que inicializa todos los atributos usando los parámetros correspondientes, excepto el iterador, que se establecerá mediante la llamada al método `setItList`, que también debes definir (observa que cuando se construye el objeto aún no se dispone del iterador de su posición en la lista, y por tanto éste tomará valor mediante la llamada a dicho método). Define también una constructora que construye el objeto leyéndolo de un flujo `ifstream`.

Clase Pacman: Hereda de la clase `GameCharacter`, define las constructoras necesarias (delegando la construcción de la parte heredada en la constructora correspondiente de su clase base), añade los atributos y métodos extra necesarios, implementa el método `update` (más los métodos auxiliares necesarios) y re-define el método `saveToFile` para incorporar la escritura de los atributos añadidos según corresponda.

Clases Ghost y SmartGhost: La clase `Ghost` hereda de `GameCharacter`, define las constructoras necesarias e implementa al menos el método `update` con el movimiento aleatorio básico de los fantasmas simples. La clase `SmartGhost` hereda de `Ghost`, añade al menos un atributo para manejar la edad del fantasma, y redefine los métodos `update` y `saveToFile` (haciendo uso de los métodos correspondientes de la clase base).

Clase GameMap: Hereda de la clase `GameObject`, implementa el método `render` y añade respecto a la versión de la práctica 1 la implementación del nuevo constructor que recibe un flujo `ifstream` y del método `saveToFile`. Se recomienda implementar un método que dado el rectángulo `SDL_Rectangle` correspondiente a un objeto determine si éste tiene intersección con un muro del mapa.

Clase Game: Añade el soporte necesario para llevar a cabo las nuevas funcionalidades indicadas. Además, ahora los objetos del juego deben guardarse en una lista polimórfica (tipo `list<GameObject*>`) y por tanto los recorridos y búsquedas sobre ella deben hacer uso de iteradores. Es conveniente guardar una segunda copia de los punteros a los distintos objetos de manera explícita y separada, como el mapa, el pacman o los fantasmas, en este caso mediante otra lista (tipo `list<Ghost*>`). Finalmente, como se explica más abajo, incluye una lista con los iteradores a los objetos que corresponde destruir al acabar el bucle de actualizaciones.

Eliminación dinámica de objetos

Ciertos objetos (en particular los fantasmas evolucionados) pueden saber que deben dejar de existir en su método `update`. En tal caso se invocará al método `eraseObject` del juego quien llevará a cabo el borrado del objeto. Para no borrar al objeto mientras éste sigue ejecutando su método `update` (lo que podría fácilmente ocasionar errores de accesos no válidos a memoria), el juego guardará una lista con las posiciones de los objetos (como iteradores) que deben borrarse en cuanto acabe el bucle de actualizaciones. Para evitar búsquedas innecesarias, cada objeto del juego guardará un iterador a su posición en la lista de objetos, que le pasará como parámetro a la llamada a `eraseObject`, y que precisamente se almacenará en la lista mencionada para su posterior borrado.

Jerarquía de excepciones

Debes implementar al menos las siguientes clases para manejar excepciones:

PacmanError: Hereda de *logic_error* y sirve como superclase de todas las demás excepciones que definiremos. Debe proporcionar por tanto la funcionalidad común necesaria. Reutiliza el constructor y método `what` de `logic_error` para el almacenamiento y uso del mensaje de la excepción.

SDLError: Hereda de `PacmanError` y se utiliza para todos los errores relacionados con la inicialización y uso de SDL. Utiliza las funciones `SDL_GetError`, `IMG_GetError` y `TTF_GetError` (si corresponde) para obtener un mensaje detallado sobre el error de SDL que se almacenará en la excepción.

FileNotFoundError: Hereda de `PacmanError` y se utiliza para todos los errores provocados al no encontrarse un fichero que el programa trata de abrir. El mensaje del error debe incluir el nombre del fichero en cuestión.

FileFormatError: Hereda de `PacmanError` y se utiliza para los errores provocados en la lectura de los ficheros de datos del juego (fichero de partida guardada, y en caso de usarlo, fichero de nivel). Debes detectar al menos dos tipos de errores de formato.

Guardado/carga de partidas

Los ficheros (en formato texto) con partidas guardadas empezarán con la información propia del objeto `Game` (el nivel, la puntuación, etc), viniendo a continuación el número de objetos de la escena y secuencialmente cada objeto (mapa, fantasmas y pacman), opcionalmente en un orden establecido, cada uno con su formato correspondiente (con sus partes comunes de acuerdo a la jerarquía de objetos). La lectura de cada objeto se realizará mediante la llamada al constructor correspondiente con el flujo ya abierto, y la escritura mediante la llamada polimórfica al método `saveToFile` del objeto en cuestión.

Pautas generales obligatorias

A continuación se indican algunas pautas generales que vuestro código debe seguir:

- Se debe hacer un buen uso de la herencia y del polimorfismo de manera que el código sea claro, se eviten las repeticiones de código, distinciones de casos innecesarias, y no se abuse de castings ni de consultas de tipos en ejecución.

- Asegúrate de que el programa no deje basura. Para que Visual te informe debes escribir al principio de la función `main` esta instrucción

  `_CrtSetDbgFlag( _CRTDBG_ALLOC_MEM_DF | _CRTDBG_LEAK_CHECK_DF );`

  Para obtener información más detallada y legible debes incluir en todos los módulos el fichero `checkML.h` (disponible en la plantilla en el proyecto `HolaSDL`).
- Todos los atributos deben ser privados/protegidos excepto quizás algunas constantes del juego en caso de que se definan como atributos estáticos.
- Define las constantes que sean necesarias. En general, no deben aparecer literales que pudiesen corresponder con configuraciones del programa en el código.
- No debe haber métodos que superen las 30-40 líneas de código.
- Escribe comentarios en el código, al menos uno por cada método relevante que explique de forma clara qué hace el método. Sé cuidadoso también con los nombres que eliges para variables, parámetros, atributos y métodos. Es importante que denoten realmente lo que son o hacen. Preferiblemente usa nombres en inglés.

Funcionalidades opcionales (2 puntos adicionales máximo)

- Premios: Cuando el pacman se come a un fantasma, con cierta probabilidad, aparece un premio en una posición del mapa (por ejemplo en el lugar en el que en ese momento se encuentre un fantasma) durante un tiempo, después del cual se eliminará de la escena. Si el pacman coge el premio se aplicará su acción correspondiente. Algunos ejemplos de acciones asociadas a premios pueden ser: obtener una vida, pasar de nivel, incrementar (durante un tiempo) la velocidad del pacman, etc.
- Implementa el soporte necesario para que el menú inicial se gestione desde la propia ventana SDL y que no haya que ir a la consola para ninguna acción (incluyendo la lectura del código numérico de los ficheros).
- Utiliza el paquete `TTF` de SDL para manejar los distintos textos (incluyendo el contador) que se utilizan en el juego. Se darán detalles en clase.

Entrega

En la tarea del campus virtual *Entrega de la práctica 2* y dentro de la fecha límite (ver junto al título), uno de los miembros del grupo, debe subir un fichero comprimido (.zip) que contenga la carpeta de la solución y el proyecto limpio de archivos temporales (asegúrate de borrar la carpeta oculta .vs y ejecuta en Visual Studio la opción "limpiar solución" antes de generar el .zip). La carpeta debe incluir un archivo `info.txt` con los nombres de los componentes del grupo y unas líneas explicando las funcionalidades opcionales incluidas y/o las cosas que no estén funcionando correctamente. Ese mismo texto debes subirlo también en el cuadro de texto (sección "texto en línea") asociado a la entrega.

Además, para que la práctica se considere entregada, deberá pasarse una *entrevista* en la que el profesor comprobará, con los dos autores de la práctica, su funcionamiento en ejecución, y si es correcto realizará preguntas (posiblemente individuales) sobre la implementación. Se darán detalles más adelante sobre las fechas, forma y organización de las entrevistas.